Modularity and Enterprise Software

Komal Gohil, A.Nagalakshmi
Innovation & Research Labs, Patni Computer Systems Ltd., Seepz, Andheri(E) Mumbai 400 096.
komal.gohil@patni.com, nagalakshmi.alapati@patni.com

***Abstract*—Modern Enterprise Software Systems (MESS) is all about envisioning, developing, managing and evolving enterprise applications to fulfill business requirements. This may entail many challenges like rapidly changing business scenario, increase in complexity, shorter time to market and business agility. In order to deal with this natural evolution, achieving modularity across MESS is essential. In this paper, we describe by way of an example application, some of the common problems encountered while delivering & managing enterprise software. We demonstrate that one of the root causes for these is inadequate support for modularity at the physical level viz. packaging & deployment. We look at the different options available for extending the modularity across packaging and deployment e.g. Impala and Open Service Gateway initiative (OSGi). Based on our explorations and experiments we provide a comparison between the two. We conclude the paper with a note on the future directions for physical modularity.**

I. CHALLENGES FACED BY ENTERPRISE SOFTWARE

Evolution of business has led IT make breakthrough progress. The software architecture has grown from monolithic to client-server to web-based applications. The web based applications and services have given rise to Modern Enterprise Software Systems (MESS) which are growing beyond the boundaries of silo-ed enterprise applications. They are getting integrated within and across the enterprises thereby evolving into huge ecosystems like Google Search, Amazon, eBay, etc. These systems are being used by multiple people/systems across multiple geographical locations and round the clock. Managing and maintaining MESS is a challenge in itself.

Let us understand the problems faced while delivering & managing MESS with an illustrative example.

Banquet Hall Reservation application, based on Spring Faces i.e. Spring and JavaServerFaces technology allows following functionality:

- Search of available rooms
- Facilities available in the respective room
- Registering user and profile
- Booking Service for room booking, Updating details, Cancellation of the booking
- Amenities – like sea view, top floor etc.

The entire code is bundled under the package structure *org.springframework.webflow.samples.booking*. As shown in Fig.1. The java code is available at *src/main/java*, the database related import files are available at *src/main/resources*. The web pages are kept in the *src/main/webapp/WEB-INF/flows.*

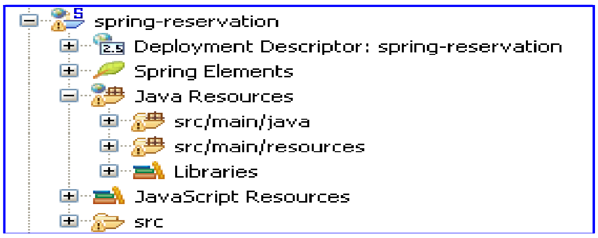

Fig.1: Application Structure of Banquet Hall Reservation in Spring

On compiling, all the java class files get into *src/main/webapp/WEB-INF/classes.* For deploying we end up having one directory called *src.* If we do changes in a particular module the entire application needs to be deployed again. The server needs to be restarted so that the new changes get picked up.

If there are multiple teams working on different modules, deploying the whole application is not an ideal case. It is expected that each team is as independent as possible so that there is minimal impact on their work. With the ever-increasing complexity of MESS, following such a deployment mechanism makes the application prone to errors. If we need to allow frequent updates, debugging and testing each unit separately, then we need to look out for a better modularity solution which will allow us to keep separate things separate and related things together.

Continuous integration, robust systems, scalable development – are some of the mandates of MESS. Seeing the above way of application structure making these goals truly workable is something which we need to look out for.

II. COMMON RECURRING PROBLEMS

Often when we feel that certain functionality has been well implemented, it may go for a toss when we move to production. We are not sure whether our application will behave the same way when it goes live. Some of the common problems that we face are:

- Unwanted interactions – During deployment we come across cases which lead to unwanted and unintentional interactions across different modules. This may arise if we happen to put a jar with the same name that is also referred by a different module.
- Integrations with other applications – Integration points are one of the high alert areas, which may lead to system failure. A change in the signature of an API that is being referred may lead to exceptions.
- Cascading failures – Failures at packaging level may lead to failures at development and design level as well as management and monitoring level.

DOI: 01.IJIT.01.01.69

- Failures in deployment - Application which seem to work fine in a development box fail to make it to operational environment owing to package version problems or other dependencies.
- Chain reactions – If in a load balancer condition, one of the servers fail, the load is passed on to others. There is high possibility that the remaining servers may succumb to the extra load and in turn fail.
- Blocked Threads – Blocked threads may also lead to chain reactions. This happens when the request-handling threads in an application get blocked and the application stops responding.

These problems are well narrated in Michael Nygard's book called 'Release it' [1] and Michael Russell's series of articles on quality busters [2].

Solving these problems becomes a mandate for us as an enterprise application has to exist within an operational environment, which is the life-support system for the application. Therefore apart from targeting needs from the business community we also have to focus on providing end-to-end modularity across the application. We need to understand the causes for the above failures so that such surprises are taken care well in advance.

III. ROOT CAUSE: INSUFFICIENT MODULARITY

Java provides modularity by way of convention. At the file level, it provides the concept of package construct and class, which help in achieving logical modularity. JARs help us in bundling these classes and thereby providing physical modularity. Building JARs is left to the responsibility of the developer and managing them is again a big challenge. We often come across classloading problems which lead to exceptions like –

- Class not found Exception – This is a very common error that we face if the class is not made available in the global classpath.
- Conflicting classes – If there are two classes with the same name there is a high likely that we may end up using an obsolete class.
- Version dependency problem – With continuous deployment happening there is high chance that new version of a jar gets released every now and then. But it is not always possible to point to the latest jar as we may not be ready with the changes. In such cases we need to have version specific dependency.
- Entire JAR is exposed – When a jar is deployed, all those classes declared public are accessible to clients outside the JAR which we may not want to. This is another symptom of the lack of any runtime modularity across JAR files.

Similarly at the design level, Java Enterprise Edition (JEE) provides us with servlet container, EJB container etc. By way of this partition we are able to segregate the different layers of code. Servlets can be contained in the servlet container, the EJBs in the EJB Container, database procedures in the database server. But there is no provision for managing modularity across servlets or EJBs etc within their respective containers.

Spring has taken care some of these problems at the design and development level. Spring provides us applicationContext.xml file where in we can provide details of the dependencies i.e. define business logic beans, resources, and all other beans that are not directly related to the web tier. As the applications grows the size of this file also increases which becomes hard to manage and maintain. Though there are workarounds for splitting the file, yet they do not provide an effective mechanism of modularity. Spring also provides auto-wiring configuration for dependency management which once again becomes difficult to manage if the application is growing in leaps.

Generally when we start with the development work, we split the entire functionality into different modules and each team is assigned a specific module to work on. But when it comes to deployment, we deploy it as one big monolithic WAR file. The modularity which was maintained so far gets lost with this nature of packaging and deployment.

In order to achieve modularity across application from design to management and monitoring, we need conceptual and logical modularity coupled with physical modularity. Java gives packaging and JAR facility but then we end up having class loader problems. Thus the basic facilities provided by Java and Java EE help us to achieve modularity at design and development level. Though these disciplines are necessary, they are not sufficient for building & managing a sizable enterprise software system. The modularity needs to be carried forward to the packaging, deployment, management, and monitoring level.

All these imperatives lead to the fact that we need to look at an alternate mechanism other than Spring which allows us to fortify the boundaries of modules such that outsiders can see and use only the published API of the library. Lets looks at solutions like - Impala and Open Service Gateway Initiative (OSGi) have in store for us.

IV. SOLUTION 1: IMPALA

Impala [5] is a dynamic module framework for Java based web applications, which is built on Spring Framework. It allows us to divide a large Spring-based application into a hierarchy of modules. These modules can be dynamically added, updated or removed. This allows us easier maintenance and management of modules.

Impala also provides build system, based on ANT. With its *ant newproject* command, application is auto magically modularized into build (host) module, main (root Impala) module, non-root module, repository module, test module and web module.

Using Impala framework, we have re-structured the same Banquet Hall Reservation application. Fig.2 provides the details of the application structure.

- BHallR-build – It comprises of the build file.
- BHallR-dataaccess – It has the database related files.

DOI: 01.IJIT.01.01.69

- BHallR-host – This will have the context folder which contains the web context root, impala libraries
- BHallR-repository – Contains a local repository of the third party jars used in the application. This directory contains a number of subdirectories, making it easier to specify which jars to include when assembling distribution archives.
- BHallR-service – Another child of the root module which contains the data access logic. There isn't really a distinct service versus data access layer in the Spring application.
- BHallR-tests – This project is purely present as a location for running test suites. It allows us to run suites or individual tests in any of the projects.
- BHallR-web – A module which implements the presentation layer for the Banquet Hall Reservation application.
- BHallR-main – This is the place where we can run the application

Modularity across packaging and deployment is very much visible. Here each module can be deployed & un-deployed independently.

New modules can be added with command *ant newmodule* command in the application. Impala dependencies can be obtained with *ant fetch* command. Fig.3 shows the dependencies.txt in the main project. Similarly the third party library dependencies can be obtained from the Maven repositories.

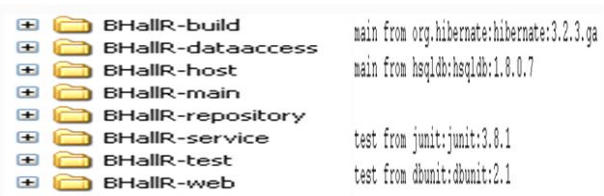

Fig 2: Structure with Impala Framework Fig3.dependencies.txt

A. Features of Impala:

Following are the features of Impala:

- We can specify a complex Spring application as a set of modules fitting into a module hierarchy.
- Modules can be dynamically added, removed and updated. Class changes are automatically reflected when modules are reloaded.
- We can develop and test our application without having to leave Eclipse. In other words, we can do most of our development using a completely build free development/deploy/test cycle.
- We can run Spring integration tests interactively. For example, we can:
  - rerun modified tests without having to reload Spring application contexts
  - reload individual modules without having to restart JVM or reload full Spring context
  - reload full application context without having to restart JVM or reload third party libraries.
- We can also run Spring integration tests in a traditional way: via the ANT Junit task, via Eclipse, both individually or as a suite.
- With the help of Jetty, we can run a web application without having to build or deploy.
- Web applications can equally be deployed as WAR files and run in Tomcat (or other application servers) with little effort.
- Impala web applications can be single- or multi-module, in the latter case with a separate module per servlet.
- We can manage dependencies by using Impala's download task to retrieve our projects directly from one or more local or remote Maven repositories.

B . Limitations of Impala:

Following are the limitations of Impala based on the implementation that we carried out:

- In traditional enterprise the application is loaded in a single classloader, which may lead to conflicts across class files. Impala takes a different approach. The classloader and spring context are loaded per module but the third party libraries are loaded in the same way as in standard standalone or web applications. This means that we cannot have two different versions of the same libraries in the application at a time. i.e. It follows the graph based classloading approach for application modules but hierarchical based classloading approach for third party libraries.
- The third party libraries which are required must be specified in a "dependencies.txt" file and must follow a particular format like main from commons-logging: commons-logging:1.1. This indicates that we want the Maven repository jar file with organization id commons-logging, artifact id commons-logging and version number 1.1. to be downloaded and placed into the workspace's repository folder main.
- Impala does not provide any IDE tool support.

Due to the limited support for third party libraries and IDE tool support, let's look into the second alternative, which is Open Service Gateway initiative.

V. SOLUTION 2: OPEN SERVICE GATEWAY INITIATIVE (OSGI)

OSGi [6] [7] is a specification, which provides applications to change the composition dynamically without requiring restart and allows them to run on shared environment. OSGi provides a service registry that enables these components to dynamically discover each other for collaboration. OSGi extends JAR's physical modularity into a logical component model known as bundle.

A. Framework – Layered Architecture

The OSGi framework implements a complete and dynamic component model. Applications or components can be

DOI: 01.IJIT.01.01.69

remotely installed, started, stopped, updated and uninstalled without requiring a reboot. The service registry allows bundles to detect addition of new services, or the removal of services, and adapt accordingly. Fig.4 shows OSGi framework as a layered architecture composed of four layers.

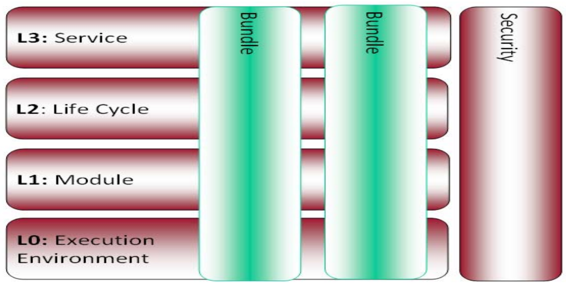

Fig.4: OSGi Framework – Layered Architecture

- L0: Execution Environment – OSGi minimum execution environment
- L1: Modules – Defines encapsulation and declaration of dependencies
- L2: Life Cycle Management – Manages the lifecycle of a bundle without requiring the VM to be restarted
- L3: Services Registry – Provides publish/find/bind model to decouple bundles

B. Bundles

In OSGi, software is distributed in the form of bundle, a unit of modularization. Bundle comprises of JAR file along with metadata added in the form of additional headers in the MANIFEST.MF file (Fig.5). Bundles may “export” and “import” packages to/from other bundles and/or may provide/use services to/from other bundles.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: DateService Plug-in
Bundle-SymbolicName: DateService
Bundle-Version: 1.0.0
Bundle-Activator: pace.osgi.demo.dateservice.impl.Activator
Import-Package: org.osgi.framework;version="1.3.0"
Export-Package: pace.osgi.demo.dateservice
Bundle-Vendor: Komal Gohil @ Patni
```

Fig.5: OSGi manifest.mf File

C. OSGi enables Modularity through Static, Dynamic and Declarative

a. Static Modularity – In OSGi, modules can be built and deployed as separate JARs. It is possible to have multiple versions of a bundle in an OSGi system at once. This might be useful if we have an old client, which depends on older version of our API, and at the same time there are others which depend on the latest version.

b. Dynamic Modularity – Bundles are isolated in their own class space so that any given bundle may be installed, stopped, started, updated, or uninstalled independently of other bundles in the framework there providing dynamic modularity. This makes it possible to swap a bundle with a newer version of the same bundle while the application continues to run.

c. Declarative Modularity – Given that services can dynamically come and go at runtime, clients should be able to cope when the service isn’t present. In order to manage this we have a declarative approach where we can define components and then hook them up together, without having a programmatic dependency on the OSGi APIs.

D. OSGi with Spring – Spring Dynamic Modules

OSGi complements Java EE as it augments Java packaging and deployment model. It eliminates the concept of monolithic EAR & WAR file. The applications can now be deployed as a set of OSGi bundles. This solves the problems related to classpath.

OSGi combines with spring called Spring Dynamic Modules (Spring DM) [8] [9] to bring the benefits of Modularity, Versioning, Hot Deployments, Service Orientation, Dependency Management and et al. to Spring-based enterprise application development.

Spring DM provides simplicity of Plain Old Java Objects (POJOs) to OSGi. It provides a spring configuration namespace that enables us to declare and publish Spring beans as OSGi services. This allows us to consume OSGi services as if they were just beans in a Spring application context.

The Banquet Hall Reservation application was re-organized using Spring DM Framework. Though the front-end of the application will not alter as compared against Impala but the way the applications are structured will change, as shown in Fig.6.

com.patni.bhall.reservation
com.patni.bhall.reservation.resource
com.patni.bhall.reservation.search
com.patni.bhall.reservation.webapp

Fig. 6: Project Structure in Spring DM

The application is split module wise viz. bundles, based on the functionality. The search related files were put under com.patni.bhall.reservation.search. The front-end pages and the reservation related files were kept in *com.patni.bhall.reservation.webapp*, *com.patni.bhall.reservation* and *com.patni.bhall.reservation.resource* were kept for future extension of the application. All these modules have respective context files as well as specific test folders for running separate unit tests. Each bundle has a src and target folders. Target includes the class files, src will be very similar to the standard spring web-app which we saw in the beginning.

Thus each bundle is independent and acts like a full-fledge application as compared against the initial Spring application. With the availability of separate bundles, it becomes easy to deploy only the specific module which are undergoing changes and keep the rest of the modules intact.

Thus each module can be deployed & un-deployed independently.

In the case of Impala, the application had a standard structure where as in OSGi it has been modularized based on the way the bundles were created.

DOI: 01.IJIT.01.01.69

VI. Comparison: Spring Dynamic Modules (OSGi) vs Impala

Both the frameworks aim for achieving modularity at run-time for spring based applications. If we have an OSGi compliant environment, it may be ideal to go for Spring Dynamic Modules. On the other hand if there are less third party dependencies, then Impala may be a better choice. On similar ground, based on our explorations, experiments and experience we have put forward a comparative analysis between Spring Dynamic Modules and Impala. (Refer Table 1.0)

OSGi is a well-established framework. The need for modularity in Java EE, has brought OSGi into picture and that is how Spring Dynamic Modules have come in. In other words OSGi is complete from the ground up Java modularity solution. But as it caters to so many needs it has become complex.

On the other hand, Impala is a less complex module framework. Impala was evolved specifically to handle large Spring-based projects. So it has gone in a top-down fashion and hence it is yet to handle the limitations on the usage of third party libraries. It works with other frameworks like Struts, Wicket, JSF etc. It does not depend on any third-party runtime environments.

Table I:
Comparison between Spring DM and Impala

Characteristics	OSGi/Spring Dynamic Modules	Impala
Structure App. in Modular way	Yes	Yes
Dynamic Reloading	Yes	Yes
Select part to Deploy	Yes	Yes
Web App. using Spring MVC	Yes	Yes
IDE plug-in	Available	There is no IDE plugin available for using Impala.
Environment	Any OSGi R4.0 compliant container	Tomcat, Jetty
Classloading Policies	Graph based	Graph + Old traditional Java (Hierarchical based)
Runtime requirements	JVM 1.4, OSGi container	JVM 1.5
Deployment options	OSGi compliant bundles	In file system, standalone jars, Jars in war files
Build	Maven	ANT based build
Support for multi-language applications, with different modules built in alternative languages	Yes	No
Load two different versions of the same library	Yes	No, as the third party libraries follow the traditional way of classloading policies.
Dependency Management	This happens through - Import/Export Package; Required bundles/Libraries	This is managed by the dependencies.txt file
Stable release	Yes	1.0 RC4 has been released. The final version of release 1.0 is due shortly.
Industry Adoption	Became a de-facto standard. e.g. Ecllipse	Impala is an upcoming initiative and is yet to get adopted.

VII. Future Directions

Modularity is critical for managing the complexity and maintaining the quality of large enterprise software systems. So far we have seen modularity at packaging and deployment level. This needs to be further extended to the management and monitoring phases of MESS.

With the growing popularity of cloud computing and mobile computing, management and monitoring are becoming more important. Monitoring servers, data loads help us to be prepared for the new requirements. Similarly managing group of servers like deploying application across number of DM servers with one click is gathering momentum. In such conditions modularity becomes all the more important to ensure there are no hidden surprises.

All along we have been considering modularity across downstream activities like packaging, deployment, management, and monitoring. In order to make the day to day life with MESS more effective, we need to tie this up as a feedback mechanism for the upstream activities like design and development. These areas are potential candidates for further research and exploration. They seem promising and are likely to make huge impact on the day to day life of the people dealing with modern enterprise software systems.

Acknowledgment

The work reported here has been carried out under the guidance of Mr. Sunil Joglekar, Senior Technical Architect in Innovation Research Labs at Patni Computer Systems Ltd.

References

[1] Release It!: Design and Deploy Production-Ready Software by Michael T. Nygard: http://www.pragprog.com/titles/mnee/release-it.

[2] Quality Busters by Michael Russell http://www.ibm.com/developerworks/web/library/wa-qualbust1/index.html

[3] JAR Hell: http://en.wikipedia.org.

[4] Java JAR File Specification: http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html

[5] P. Zoio, "Impala", [online]. http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html

[6] OSGi: http://www.osgi.org/Main/HomePage.

[7] Neil B., *OSGi in Practice*: http://s3.amazonaws.com/neilbartlett.name/osgibook_preview_20090110.pdf.

[8] Walls C., *Modular Java Creating Flexible Applications with OSGi and Spring*, The Pragmatic Bookshelf, 2009: http://pragprog.com/titles/cwosg/modular-java.

[9] SpringDynamicModules: http://www.springsource.org/osgi

About the Author

Komal Gohil is a Software Engineer in Innovation Research Labs at Patni Computer Systems Ltd. She had carried out this work as Research Trainee with Innovation Research Labs at Patni Computer Systems Ltd. as a part of accomplishment of her M.Tech in Sardar Vallabhbhai National Institute of Technology, Surat (Gujarat).

A.Nagalakshmi is a Manager in Innovation Research Labs at Patni Computer Systems Ltd. She has around 13 years of experience as software professional. She has worked in diverse platforms and technologies like Mainframe/Cobol, Unix/C, Java and testing projects. Her primary areas of interest are Cloud Computing, Enterprise Mobility, DSLs et al.